**Introduzione**

Il concetto fondamentale di ogni sistema operativo è il processo: un’astrazione di un programma in esecuzione.

**Processi**

Tutti i computer moderni svolgono spesso molti compiti contemporaneamente. Per gestire tali attività è necessario un sistema di multiprogrammazione con più processi.

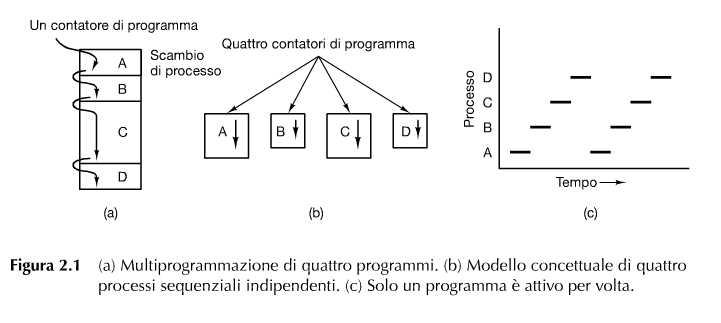
In ogni sistema multiprogrammato la CPU passa da processo a processo rapidamente, eseguendone ognuno per decine o centinaia di millisecondi, dando l’illusione di eseguire più processi contemporaneamente (pseudoparallelismo).

**Il modello di processo**

I processi sono il software che è in esecuzione o “gira” sul computer (compreso il SO). Il processo è un’istanza dell’esecuzione di un programma, inclusi i valori attuali del program counter, dello stack, dei registri e delle variabili.

Concettualmente ogni processo ha la sua CPU virtuale. In realtà la CPU passa avanti e indietro da processo a processo (multiprogrammazione).

Nella Figura 2.1(a) è mostrato un computer che fa multiprogrammazione di quattro programmi in memoria. Nella Figura 2.1(b) sono mostrati quattro processi, ognuno con il suo flusso di controllo e ognuno che gira indipendentemente dagli altri. Nella Figura 2.1(c) vediamo che, osservati su un intervallo abbastanza lungo, tutti i processi hanno avuto un avanzamento, ma in un preciso particolare istante solo un processo è realmente in esecuzione.



Mentre un programma ha una natura statica, la sua esecuzione (cioè il processo) ha una natura dinamica legata al valore del suo stato (PC, stack, ….). Un programma, inoltre, può originare più processi.

In UNIX il comando *ps* permette di conoscere i processi attivi; in Windows si utilizza il *task manager*.

**Creazione del processo**

I sistemi operativi hanno bisogno di un modo per creare processi.

Un processo viene creato durante uno di questi eventi:

- inizializzazione del sistema;

- esecuzione di una chiamata di sistema di creazione di un processo;

- richiesta dell’utente di creare un processo;

- inizio di un job in modalità batch.

All’avvio vengono in genere creati vari processi, tra cui:

- i processi attivi che interagiscono con gli utenti e svolgono un compito per loro;

- i processi in background, non associati ad un utente in particolare, che svolgono funzioni specifiche all’arrivo di una richiesta (vengono anche chiamati demoni).

In UNIX, per creare un processo nuovo esiste una sola chiamata di sistema: *fork*. Questa chiamata crea un clone esatto del processo che esegue la chiamata. Dopo la *fork* i due processi, padre e figlio, hanno la stessa immagine della memoria, le stesse stringhe di ambiente e i medesimi file aperti.

Solitamente, il processo figlio poi esegue una *execve* o una chiamata di sistema analoga per cambiare la propria immagine di memoria ed eseguire un nuovo programma.

In Windows viene utilizzato *CreateProcess()* sia per creare un nuovo processo sia per il caricamento del programma corretto nel nuovo processo.

Sia in UNIX che in Windows, il genitore e il figlio hanno spazi di indirizzi distinti: ciò significa che una modifica ad una variabile dell’uno non influenza l’altro. Invece, sono condivisi i riferimenti a medesime risorse (es file, ...).

Una volta creato, un processo comincia a girare fino a quando non ha svolto il proprio compito.

**Chiusura di un processo**

La chiusura di un processo si verifica a seguito di una delle seguenti condizioni:

- uscita normale (condizione volontaria): *exit()* in UNIX e *ExitProcess()* in Windows;

- uscita su errore (condizione volontaria): esempio bug nel programma;

- errore critico (condizione involontaria): esempio run-time error;

- terminato da un altro processo (condizione involontaria): *kill()* in UNIX e *TerminateProcess()* in Windows.

**Gerarchie di processi**

Generalmente, quando un processo ne crea un altro, il processo genitore e il processo figlio restano in un certo modo associati.

In UNIX, il processo figlio può a sua volta creare altri processi, formando una gerarchia di processi. Un processo, tutti i suoi figli e gli ulteriori discendenti costituiscono un gruppo di processi. Windows, invece, non ha il concetto di gerarchia di processi. Tutti i processi sono uguali. Quando un processo crea un processo figlio ha un token speciale (chiamato handle) che può utilizzare per controllarlo ma anche cederlo ad altri invalidando il concetto di gerarchia.

**Stati di un processo**

Sebbene ogni processo sia un’entità indipendente, con il proprio program counter e il proprio stato interno, i processi hanno bisogno di interagire con altri processi. Un processo può generare alcuni output che un altro processo utilizza come input.

Esempio UNIX: *cat chapter1 chapter2 chapter3 | grep tree*

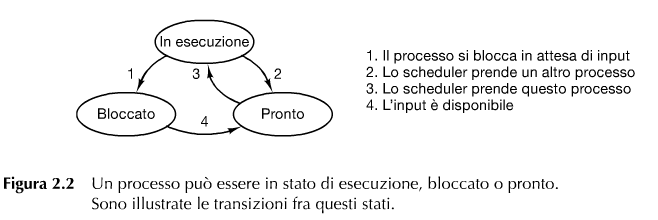
Il comando *cat* concatena tre file e ne esegue l’output. Il secondo processo, che esegue il comando *grep*, seleziona tutte le linee contenenti la parola “tree”.

Nella Figura 2.2 vediamo un diagramma di stato che mostra i tre stati in cui può trovarsi un processo:

- in esecuzione (CPU effettivamente in uso in quel momento);

- pronto (può essere eseguito, è temporaneamente sospeso per consentire a un altro processo di essere eseguito);

- bloccato (incapace di proseguire finché non avviene un qualche evento esterno).



Fra i tre stati sono possibili quattro transizioni, come mostrato nella figura. La transizione 1 accade quando il SO scopre che un processo non può continuare subito. Le transizioni 2 accade quando lo scheduler decide che un il processo in esecuzione è stato eseguito per un intervallo di tempo sufficiente. La transizione 3 si verifica quando tutti gli altri processi hanno avuto il loro equo intervallo di tempo ed è ora per il primo processo di riprendersi la CPU. La transizione 4 accade quando si verifica l’evento esterno di cui un processo era in attesa.

Un processo in esecuzione può ricevere dei segnali di allarme che, analogamente agli interrupt nel caso hardware, ne bloccano l’esecuzione, causano il salvataggio dello stato e fanno sì che il processo esegua una procedura di gestione dei segnali speciali. Al termine della procedura il processo potrà riprendere l’esecuzione dal punto dove si era interrotto.

Molto trap rilevati dall’hardware, come l’esecuzione di una istruzione illegale o di un indirizzo errato, sono convertiti in segnali per il processo responsabile.

I segnali possono essere attivati anche in modo temporizzato, come nel caso dell’invio di un messaggio per eseguire un controllo sulla ricezione entro un certo timeout.

**Implementazione dei processi**

Per implementare il modello di processo, il SO mantiene una tabella chiamata tabella dei processi, con una voce per processo, chiamata process control block. Questa voce contiene il program counter, lo stack pointer, l’allocazione della memoria, lo stato dei file aperti, le informazioni relative alla gestione e allo scheduling e qualunque altra cosa relativa al processo da salvare quando esso passerà dallo stato in esecuzione allo stato pronto o bloccato, così che possa essere riavviato in seguito come se non si fosse mai fermato.

**Modellazione della multiprogrammazione**

L’utilizzo della CPU può essere migliorato per mezzo della multiprogrammazione.

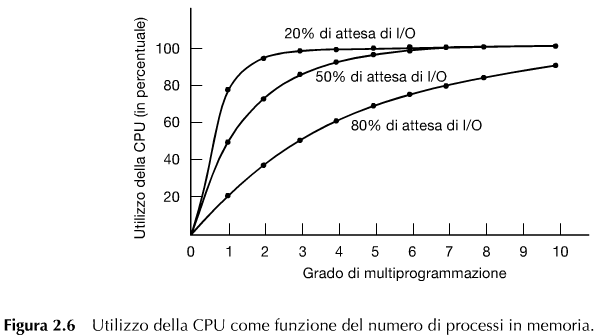
Se un processo in media esegue calcoli solo per il 20% del tempo in cui risiede in memoria, con 5 processi contemporaneamente in memoria la CPU dovrebbe essere occupata al 100%.

In realtà i processi attendono anche l’I/O.

Da un punto di vista statistico, se un processo spende una frazione *p* del suo tempo in attesa dell’I/O, con *n* processi in memoria la probabilità che stiano tutti aspettando l’I/O è *pn*.

L’utilizzo della CPU è la probabilità complementare: *1-pn*.

La Figura 2.6 mostra l’utilizzo della CPU come una funzione di *n*, chiamata grado di multiprogrammazione.



Dalla figura emerge che, se i processi impiegano l’80% del loro tempo in attesa dell’I/O, devono come minimo esserci contemporaneamente 10 processi in memoria per portare la CPU ad un utilizzo del 90%. Questo modello semplificato permette di effettuare previsioni sull’utilizzo della CPU.

Esempio: un computer ha 512 MB di memoria, se il SO occupa 128 MB e un processo 128 MB, può contenere in memoria fino a 3 processi. Se l’attesa media dell’I/O è dell’80%, l’utilizzo della CPU è pari a *1-0.83* = 49%. Aggiungendo 512MB miglioriamo del 79% = *1-0.87*.

**Thread**

Nei sistemi operativi tradizionali, ogni processo dispone di uno spazio degli indirizzi e di un singolo thread di controllo. Tuttavia ci sono frequenti situazioni in cui è auspicabile avere molteplici thread di controllo in esecuzione nello stesso spazio degli indirizzi, in quasi parallelo, come se fossero (quasi) processi separati.

**Uso dei thread**

I thread sono dei processi leggeri o miniprocessi che, a differenza dei processi, sono tra loro fortemente correlati poiché condividono spazio degli indirizzi e dati.

Esistono varie ragioni per avere questi miniprocessi, esaminiamone alcune.

In molte applicazioni ci sono molteplici attività contemporanee e, a volte, alcune possono bloccarsi. Suddividendo una di tali applicazioni in molteplici thread sequenziali eseguiti quasi in parallelo, il modello di programmazione diventa più semplice. Inoltre, essendo più leggeri dei processi, essi sono più veloci da creare e cancellare. Infine, i thread hanno un migliore utilizzo della CPU quando le attività sono I/O bound.

Un ambiente che consente a più thread di girare nello stesso processo è chiamato multithreading. Le CPU moderne che supportano il multithreading riescono a passare da un thread all’altro nell’ordine dei nanosecondi.

**Il modello a thread classico**

Il modello a processi si basa su due concetti indipendenti: raggruppamento di risorse ed esecuzione. Un processo può essere visto come un modo per raggruppare risorse relazionate. Un processo ha uno spazio degli indirizzi contenente testo, programma e dati, così come altre risorse, che possono includere file aperti, processi figli, situazioni d’allarme in sospeso, gestori di segnale, informazioni sugli account e altro.

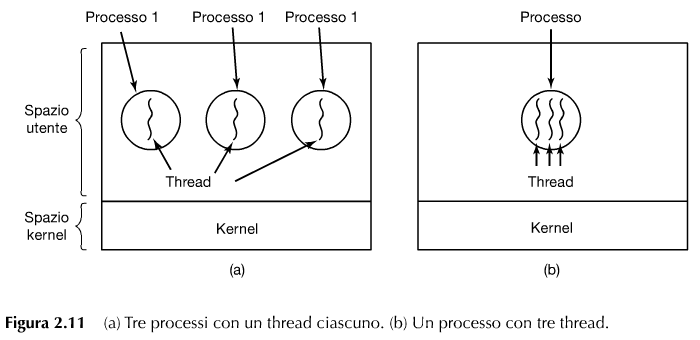
I thread invece sono entità schedulate per l’esecuzione. Più thread possono essere eseguiti nello stesso ambiente di processo.

Il thread ha un program counter (che tiene traccia di quale istruzione eseguire come successiva), alcuni registri (che contengono le sue variabili di lavoro attuali), uno stack (contenente la storia della sua esecuzione) e uno stato.

I thread hanno i medesimi stati dei processi e la stessa dinamica pronto-esecuzione-bloccato.

Più processi che girano in parallelo su un computer non è la stessa cosa di più thread in parallelo. Nel primo caso, i processi condividono solo la memoria fisica, i dischi e le altre risorse; nel secondo i thread condividono anche lo spazio di indirizzamento, lo stack, i registri e lo stato.

Nella Figura 2.11(a) vediamo tre processi tradizionali. Ogni processo ha il suo spazio degli indirizzi e un singolo thread di controllo. Nella Figura 2.11(b) vediamo invece un singolo processo con tre thread di controllo. Sebbene in entrambi i casi abbiamo tre thread, nella Figura (a) ognuno di loro opera in un diverso spazio degli indirizzi, mentre nella Figura (b) tutti e tre condividono lo stesso spazio degli indirizzi.

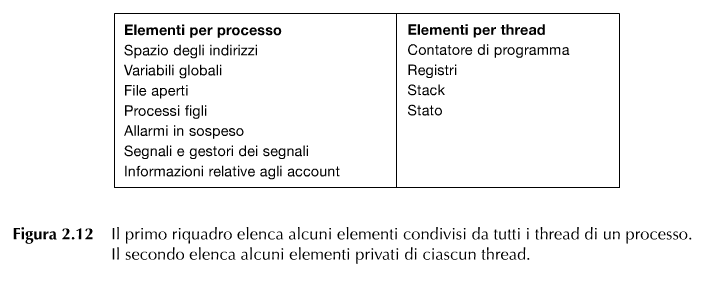


I thread in un processo non sono tanto indipendenti quanto i processi concorrenti.

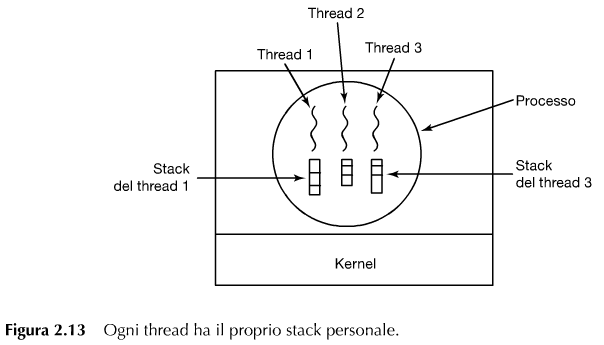
Poiché tutti i thread hanno lo stesso spazio di indirizzamento (le stesse variabili globali), possono leggere, scrivere o cancellare lo stack di un altro thread. Tra i thread, infatti, non esiste protezione perché non è possibile realizzarla e non dovrebbe essere necessaria.

Mentre i processi possono essere di proprietà di diversi utenti e competere per ottenere risorse comuni, ogni processo (di proprietà di un singolo utente) può creare più thread che dovrebbero cooperare non entrando mai in conflitto tra loro.

L’organizzazione a processi dovrebbe essere utilizzata quando le attività dei processi non sono tra loro correlate. Mentre un modello a thread si dovrebbe utilizzare quando i compiti di ciascuno sono parte dello stesso lavoro e, quindi, è necessaria una stretta collaborazione.



Ogni thread ha un proprio stack, come illustrato nella Figura 2.13.



In un ambiente multithreading i processi partono con un singolo thread, quest’ultimo ha la capacità di crearne altri chiamando la procedura *thread\_create()*. All’atto della creazione non è necessario specificare nulla poiché il nuovo thread girerà nel medesimo spazio di indirizzamento del thread che l’ha creato. Talvolta i thread possono avere relazioni gerarchiche (padre-figlio) oppure essere allo stesso livello (più comune). Il thread che crea un altro thread riceve in uscita l’identificatore del thread creato.

Quando un thread ha terminato il suo lavoro, esso può uscire chiamando la procedura *thread\_exit()*.

In alcuni sistemi, un thread può rimanere in attesa di un altro thread attraverso la procedura *thread\_join()*.

Un’altra chiamata comune è la *thread\_yield()*, che permette di rilasciare la CPU ad un altro thread. Questa chiamata è importante perché permette di realizzare la multiprogrammazione senza che ci sia interruzione, così come accade per i processi.

**Alcuni problemi presenti nei thread**

Se un processo che ha più thread crea un processo figlio, quest’ultimo avrà gli stessi thread del padre? In caso contrario, il processo potrebbe non funzionare correttamente, dato che ognuno di loro potrebbe essere essenziale. In caso affermativo, che cosa accadrebbe se un thread nel genitore fosse bloccato su una chiamata di *read*, ad esempio, della tastiera? Ora ci sarebbero due thread bloccati sulla tastiera, uno nel genitore e uno nel figlio. Quando viene digitata una riga, entrambi i thread ne riceveranno una copia? Solo il genitore? Solo il figlio?

Un’altra complicazione è relativa al fatto che i thread condividono molte strutture di dati. Che cosa accade se un thread chiude un file mentre un altro lo sta ancora leggendo?

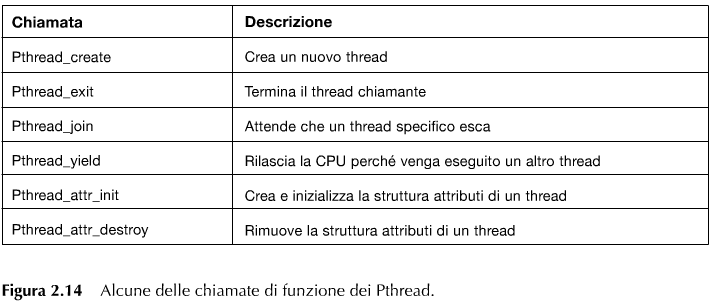
Anche se questi problemi possono essere risolti con un po’ di attenzione, occorre avere le idee chiare su come dovrebbero andare le cose nei casi descritti.

**Thread POSIX**

Per permettere la scrittura di programmi portabili che usino i thread, IEEE ha definito i thread nello standard *IEEE 1003.1c*. I thread definiti nello standard sono chiamati *pthread* e sono supportati dai principali sistemi UNIX.

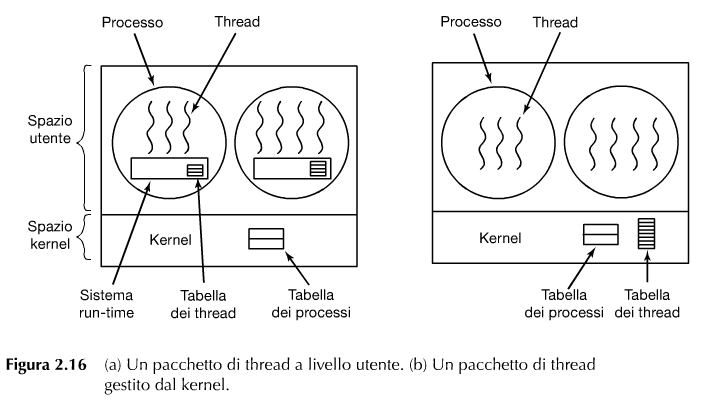
Ciascun pthread ha un identificatore, un insieme di registri (compreso il program counter) e un insieme di attributi (dimensione dello stack, parametri per lo scheduling, …) memorizzati in una struttura.

Lo standard definisce più di 60 chiamate di sistema specifiche per i pthread, alcune di esse sono illustrate nella Figura 2.14.

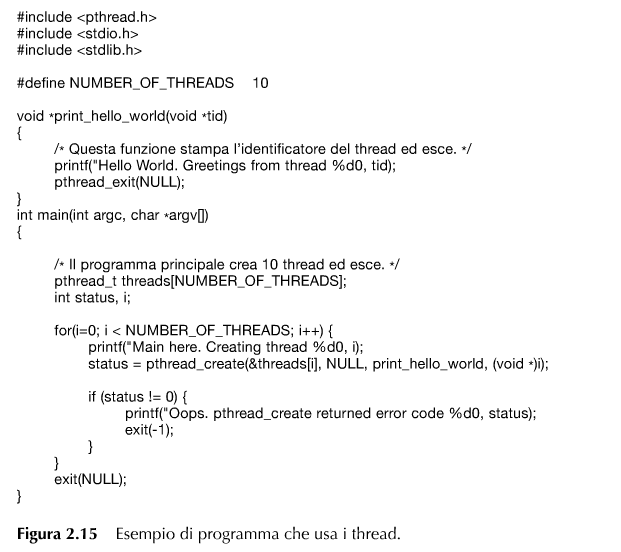


**Realizzazione dei thread**

I thread possono essere gestiti nello spazio utente (Figura 2.16(a)), nel kernel (Figura 2.16(b)) o in entrambi.



**Esempio di implementazione thread POSIX**



**Implementare i thread nello spazio utente**

Il pacchetto di thread viene messo interamente nello spazio utente. Il kernel non ne è a conoscenza poiché, per quanto lo riguarda, gestisce processi ordinari, a singolo thread.

Così facendo, è possibile realizzare thread anche su sistemi operativi che non li supportano. Normalmente sono realizzati in questo modo attraverso una libreria.

Questa realizzazione ha una struttura generale illustrata nella Figura 2.16(a). I thread sono eseguiti sopra un sistema run-time, che è una raccolta di procedure che gestiscono i thread.

In questa realizzazione ogni processo ha bisogno della sua tabella dei thread personale, per tener traccia dei thread nel processo. Essa è simile alla tabella dei processi del kernel, ad eccezione del fatto che mantiene traccia solo delle proprietà dei thread (PC, SP, registri, stato, ...). La tabella dei thread è gestita nell’ambiente di run-time.

**Vantaggi dei thread nello spazio utente**

Quando un thread termina l’esecuzione, la procedura che salva lo stato del thread e lo scheduler sono locali, in questo modo la loro invocazione è molto più efficiente rispetto ad una system call del kernel. Questo rende lo scheduling dei thread molto veloce.

Questo tipo di implementazione permette a ogni processo di avere un proprio algoritmo personalizzato di schedulazione. I thread possono crescere e replicarsi numerosi poiché non esistono dei limiti (se non quelli della disponibilità della memoria) nello spazio utente. Nel kernel invece non può crescere troppo la tabella dei processi o la dimensione dello stack.

**Svantaggi dei thread nello spazio utente**

Il primo problema riguarda l’implementazione delle chiamate di sistema bloccanti. Consentire che il thread gestisca realmente la chiamata di sistema è inaccettabile, dato che fermerebbe tutti i thread. Uno degli obiettivi principi dell’avere i thread è innanzitutto quello di permettere a ognuno di usare chiamate bloccanti, ma di impedire che un thread bloccato possa interferire con gli altri. Con le chiamate di sistema bloccanti è difficile vedere come si possa raggiungere questo scopo rapidamente.

Tutte le chiamate di sistema potrebbero essere cambiate con nuove chiamate non bloccanti oppure anticiparle con istruzioni specifiche tipo *select()*.

Un altro problema è che nessun altro thread nel processo potrà essere eseguito fino a che un thread in esecuzione non rilascia volontariamente la CPU.

Il sistema di run-time può utilizzare una modalità timesharing e round-robin utilizzando i segnali di allarme, ma oltre a creare sovraccarico al crescere della frequenza (>1 volta/sec), si corre il rischio di non poterli più utilizzare per gestire le altre situazioni dove sono comunque utili.

I programmatori usano i thread proprio nelle applicazioni che si bloccano spesso (come i Web Server che fanno molte system call), in questo caso è difficile per il kernel passare da un thread all’altro.

**Realizzazione di thread nel kernel**

Supponiamo ora che il kernel sia a conoscenza dei processi e li gestisca. La tabella dei thread è nel kernel, non nel singolo processo. Quando un thread vuole crearne uno nuovo o distruggerne uno esistente, esso esegue una chiamata al kernel, che esegue la creazione o la distruzione, aggiornando la tabella dei thread del kernel.

La tabella dei thread contiene sempre i registri dei thread, lo stato e altre informazioni, ma ora stanno nel kernel invece che nello spazio utente.

**Vantaggi dei thread nel kernel**

Tutte le chiamate che potrebbero bloccare un thread sono realizzate come chiamate di sistema. Quindi il blocco dei thread non è più un problema in quanto il kernel può passare dall’uno all’altro senza problemi. I thread nel kernel non richiedono quindi la riscrittura di chiamate di sistema non bloccanti.

**Svantaggi dei thread nel kernel**

Tutte le chiamate sono realizzate con system call, con maggiori costi rispetto ad una chiamata ad un sistema di procedure run-time.

A causa di questo elevato costo di creazione e distruzione dei thread alcuni sistemi riciclano le strutture dati utilizzate per i thread distrutti.

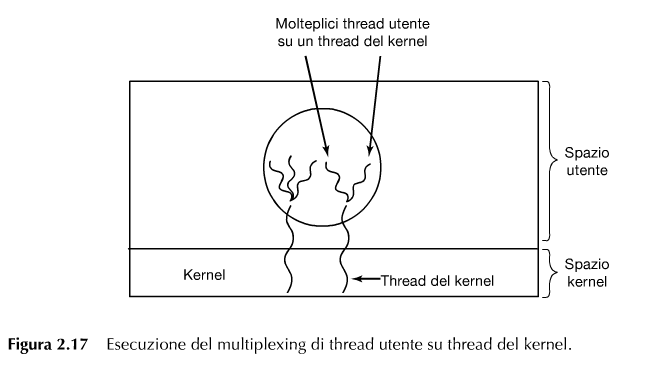
Che succede in un ambiente multithread quando un processo esegue una *fork*? Il nuovo processo ha lo stesso numero di thread del padre o solo uno?

Nel modello classico, i segnali possono essere spediti ai processi e non ai thread. Quando arriva un segnale ad un processo che contiene più thread quale di essi dovrà gestirlo?

**Implementazioni ibride**

Sono stati studiati vari modi per cercare di combinare i vantaggi delle due soluzioni.

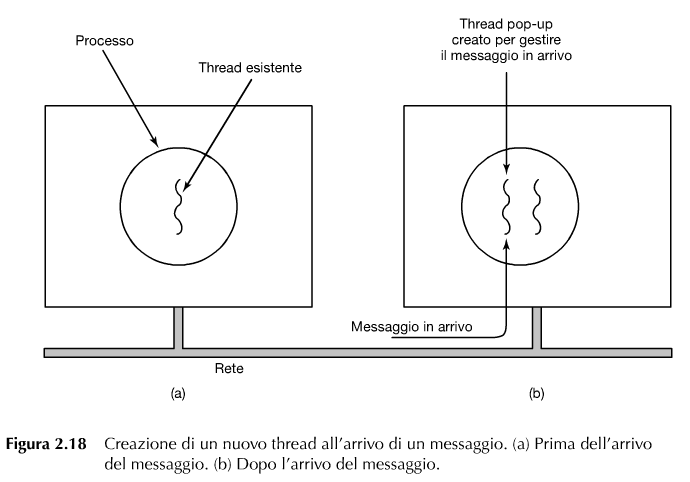
Una possibilità è usare i thread del kernel e fare in modo che ciascuno di essi utilizzi a turno un thread di livello utente, come mostrato nella Figura 2.17.



Con tale approccio il kernel è in grado di schedulare i soli thread a livello kernel.

**Thread pop-up**

I thread sono spesso utili negli ambienti distribuiti. Un esempio è la gestione dei messaggi in ingresso per richieste di un servizio. Quando arriva un nuovo messaggio il sistema crea un nuovo thread (chiamato pop-up) per gestire il messaggio stesso. Questo thread è detto thread pop-up ed è illustrato nella Figura 2.18.



Un vantaggio chiave dei thread pop-up è che, dato che sono marcati come nuovi, non hanno alcuna storia da ripristinare (registri, stack o altro). Ognuno viene avviato da zero e ognuno è identico agli altri, quindi la creazione è velocissima: quello che serve quando arriva un messaggio.

Il thread pop-up può essere eseguito sia nello spazio utente che nel kernel. Di solito, si preferisce questa seconda strada per la semplicità di accesso alle tabelle del kernel e ai dispositivi di I/O, anche se è più pericolosa.

**Comunicazione fra processi**

I processi necessitano frequentemente di comunicare con altri processi, preferibilmente in un modo ben strutturato, non usando gli interrupt. Questa comunicazione tra processi si chiama IPC (InterProcess Comunication).

I problemi da affrontare sono tre:

- come un processo possa passare informazioni a un altro;

- essere sicuri che due o più processi non vadano ad accavallarsi;

- avere la corretta sequenzialità quando vi sono delle dipendenze.

È anche importante menzionare che due di questi problemi si possono applicare ugualmente bene anche i thread. Il primo invece è semplice per i thread, dato che condividono uno spazio degli indirizzi comune.

**Race condition**

I processi che stanno lavorando insieme possono condividere una parte di memoria comune che ciascuno può leggere e scrivere.

Le corse critiche nascono quando due o più processi stanno leggendo o scrivendo dati condivisi e il risultato finale dipende dalla sequenza di esecuzione.

**Regioni critiche**

La chiave per evitare le corse critiche è vietare che più processi possono leggere e scrivere le risorse condivise contemporaneamente. In altre parole, abbiamo bisogno di una mutua esclusione, ossia un qualche sistema per essere certi che, se un processo sta usando una risorsa, l’altro processo venga escluso dal fare la stessa cosa.

La scelta delle operazioni primitive appropriate per il raggiungimento della mutua esclusione è uno dei maggiori problemi progettuali di ogni sistema operativo.

La questione delle race condition può anche essere formulata nel seguente modo: un processo è, per parte del tempo, impegnato in calcoli interni che non portano a race condition. Tuttavia, un processo deve talvolta accedere alla memoria condivisa o svolgere attività critiche che possono portare alla race condition. La parte di programma in cui si accede alla memoria condivisa è chiamata regione critica o sezione critica. Se possiamo fare in modo che due processi non si trovino mai nelle loro regioni critiche allo stesso momento, avremmo trovato la soluzione alle race condition.

Per ottenere una buona soluzione servono quattro condizioni:

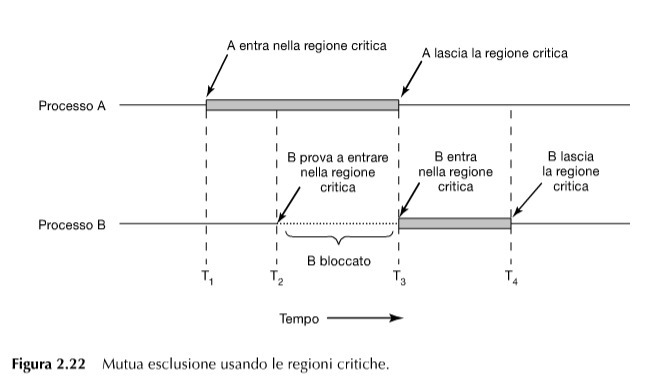
- due processi non devono mai trovarsi contemporaneamente nelle loro regioni critiche;

- non può essere fatto alcun presupposto sulle velocità o sul numero delle CPU;

- nessun processo in esecuzione al di fuori della sua regione critica può bloccare altri processi;

- nessun processo dovrebbe restare in attesa di entrare nella sua regione critica per sempre.

Il comportamento che si vorrebbe è illustrato nella Figura 2.22.



**Mutua esclusione con busy waiting**

La mutua esclusione è possibile utilizzando il busy waiting: mentre un processo è nella sua regione critica gli altri devono attendere.

Esistono varie soluzioni.

**Disabilitare gli interrupt**

In un sistema a singolo processore la soluzione più semplice è che ogni processo disabiliti tutti gli interrupt appena entrato nella sua regione critica e li riabiliti all’uscita. In questo modo la CPU non può passare da un processo all’altro.

Ogni processo può aggiornare le risorse condivise senza timore che qualsiasi altro processo possa interferire.

Un simile approccio non è generalmente molto invitante perché non è saggio dare ai processi utente la possibilità di disabilitare gli interrupt. Di solito è il kernel che decide quando disabilitare gli interrupt se sta eseguendo istruzioni critiche (in questo caso un processo utente che si intromette può portare a degli errori).

Inoltre, se il sistema è multiprocessore, disabilitare gli interrupt interessa solo la CPU che ha eseguito l’istruzione di *disable*. Le altre continuerebbero l’esecuzione e potrebbero accedere alla memoria condivisa.

Questa non rappresenta una buona soluzione, sono necessari sistemi più sofisticati.

**Variabili lock**

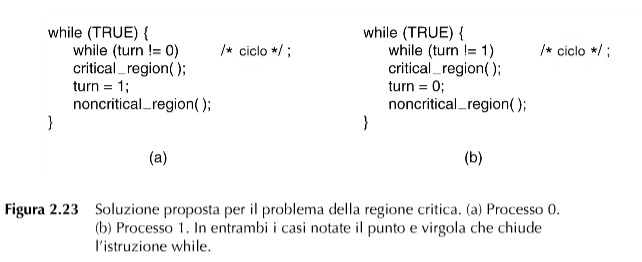
Si tratta di una soluzione software semplice: si utilizza una variabile condivisa (chiamata lock), inizialmente posta a 0. Quando un processo vuole entrare nella sua regione critica, prima controlla il lock. Se esso è 0, il processo lo imposta a 1 ed entra nella regione critica. Se invece il lock è posto a 1, il processo aspetta finché non vale 0.

Questa soluzione presenta però il seguente difetto: supponiamo che un processo legga il lock e veda che è 0. Prima che riesca a metterlo a 1, viene eseguito un altro processo e, anch’esso leggendo il lock a 0, lo imposta ad 1. A questo punto sono entrambi nella sezione critica.

Non vi è alcuna garanzia che una risorsa ritenuta “libera” perché ha lock posto a 0 non diventi occupata ancora prima che il processo le assegni il valore di “occupata”.

**Alternanza stretta**

Un terzo approccio della mutua esclusione è mostrato nella Figura 2.23.



La variabile interna *turn*, inizialmente a 0, tiene traccia di a chi tocca entrare nella regione critica. In principio, il processo 0 controlla *turn*, la trova a 0 ed entra nella sua regione critica. Anche il processo 1 crede che sia 0 e quindi rimane in attesa, in un rapido ciclo, testando continuamente *turn* per vedere quando diventa 1.

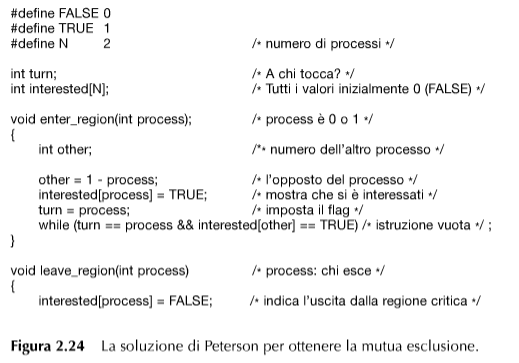
Quando il processo 0 lascia la regione critica, imposta *turn* a 1, per permettere al processo 1 di entrare nella sua regione critica.

Questa tecnica è chiamata busy waiting e andrebbe evitata perché spreca tempo di CPU. Si dovrebbe utilizzare quando l’attesa è breve. Un lock che utilizza il busy waiting è chiamato spin lock.

Questo algoritmo evita le corse critiche ma viola la condizione 3, cioè che un processo possa rimanere bloccato se un altro non è nella regione critica.

**Soluzione di Peterson**

L’algoritmo di Peterson è mostrato nella Figura 2.24.



Prima di entrare nella propria regione critica, ogni processo chiama *enter\_region* con il suo numero di processo personale come parametro. Questa chiamata farà si che si metta in attesa, se necessario, finché non è sicuro entrare. Dopo che ha finito con la regione critica, il processo chiama *leave\_region* per indicare che ha terminato e per consentire eventualmente l’ingresso dell’altro processo.

Anche se i due processi invocassero quasi contemporaneamente la procedura, solo l’ultimo scriverebbe la variabile turno. In questo modo solo l’ultimo dei due avrebbe la possibilità di non rimanere bloccato dal ciclo *while*.

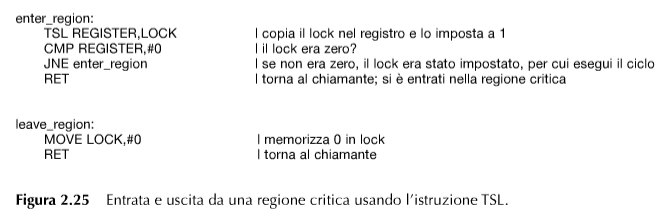
***E quindi vi è un problema di inversione di priorità***.

**Istruzione TSL**

Alcune CPU possiedono nativamente l’istruzione *TSL REGISTER, LOCK* (Test and Set Lock). Tale istruzione legge il contenuto della variabile *lock* nel registro *RX* e poi salva un valore non zero all’indirizzo di memoria *lock*. Le operazioni di lettura della parola e del suo salvataggio sono garantite come indivisibili, cioè nessun altro processore può accedere alla parola di memoria finché l’istruzione non è terminata. La CPU che esegue l’istruzione *TSL* chiude il bus di memoria per proibire ad altre CPU di accedere alla memoria finché non ha finito.

Per realizzare la mutua esclusione con questo metodo si può usare il codice illustrato nella Figura 2.25.

Prima di entrare nella regione critica, un processo chiama *enter\_region* e rimane in busy waiting finché il lock non è libero. All’uscita invoca *leave\_region*.



Un’istruzione alternativa a *TSL* è *XCHG*, che scambia il contenuto di due posizioni atomicamente, per esempio un registro e una parola di memoria.

**Sleep e wakeup**

Sia la soluzione di Peterson che l’istruzione *TSL* (o *XCHG*) sono corrette ma hanno entrambe bisogno del busy waiting (fanno sprecare tempo alla CPU). Inoltre, si consideri un computer con due processi *L*, con bassa priorità, e *H*, con alta priorità. Le regole di scheduling sono tali che *H* viene eseguito ogni volta che è in stato di pronto. A un dato momento, con *L* nella sua regione critica, *H* diventa pronto per l’esecuzione. *H* ora inizia il busy waiting, ma dato che *L* non è mai in scheduling con *H* in esecuzione, *L* non lascerà mai la sua regione critica e così *H* sarà sempre in ciclo (problema dell’inversione delle priorità).

Per superare queste problematiche esistono due primitive:

- *sleep()*: blocca il processo chiamante fino a che un altro processo non lo risveglia;

- *wakeup(p)*: sveglia il processo *p*.

**Il problema del produttore-consumatore**

Due processi condividono un buffer comune di dimensione fissata. Uno di loro, il produttore, mette informazioni nel buffer e l’altro, il consumatore, le preleva.

Esistono due problemi:

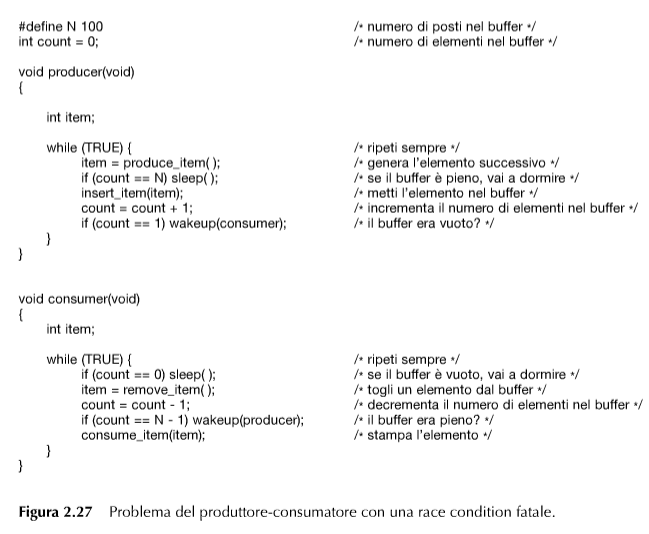
- quando il produttore vuole inserire un nuovo elemento ma il buffer è pieno;

- quando il consumatore vuole leggere un elemento ma il buffer è vuoto.

La soluzione prevede che un processo (produttore/consumatore) rimanga dormiente finché l’altro (produttore/consumatore) non inserisce/rimuove elementi.

Per tenere traccia del numero di elementi nel buffer si utilizza la variabile *count*.

Il codice è illustrato nella Figura 2.27.



Problema: supponiamo che il buffer inizialmente sia vuoto e sia avviato il consumatore che trova *count* a 0. A questo punto lo scheduler decide di interromperlo (prima della *sleep()*) e avviare il produttore. Il produttore inserisce un elemento nel buffer e incrementa *count*. Ora *count* è posto a 1, quindi cerca di svegliare il consumatore (che non dorme ancora). A questo punto lo scheduler attiva il consumatore che va subito in *sleep()*. Quando il produttore riempirà il buffer dormiranno entrambi.

La sostanza del problema è che *wakeup(p)* inviato ad un processo sveglio va perso. Per risolvere tale problematica si può aggiungere un bit di attesa wakeup. Se si invia un *wakeup(p)* ad un processo sveglio, si imposta questo bit a 1; quando il processo tenterà di addormentarsi, verificherà lo stato del bit. Se è 1 allora lo azzera e rimane sveglio; se è 0 si addormenta.

Questa soluzione termina la sua efficacia nel momento in cui aumentano i processi ed un bit non è più sufficiente a descriverne gli stati. Si possono aumentare i bit di wakeup, ma il principio del problema rimane sempre.

**Semafori**

L’idea alla base di un semaforo è quella di contare il numero di risvegli. Un semaforo può essere 0 (nessun risveglio) o assumere un valore positivo quando uno o più risvegli sono in attesa.

Su un semaforo sono possibili due operazioni: *down()* e *up()* (rispettivamente generalizzazioni di *sleep()* e *wakeup()*).

L’operazione *down()* decrementa il valore del semaforo se semaforo > 0, altrimenti (semaforo = 0) va in sleep senza completare il *down()*.

L’operazione *up()* aumenta il valore del semaforo.

È essenziale sottolineare che tutte queste operazioni vengono fatte come indivisibili azioni atomiche.

Più processi possono essere regolati da un semaforo.

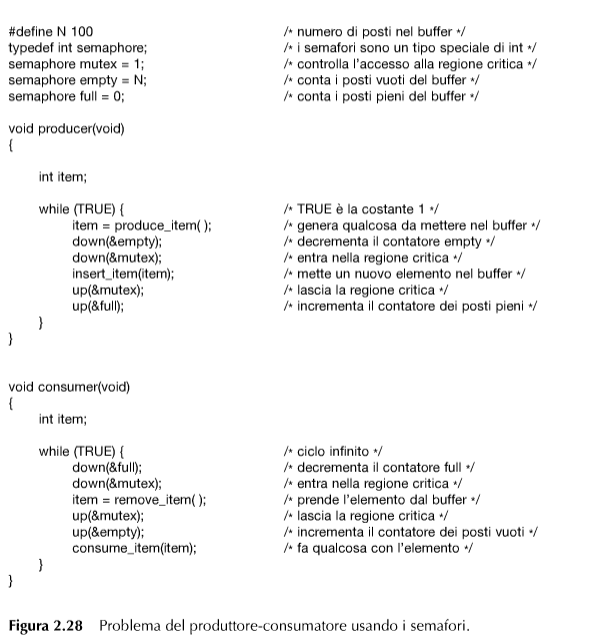
Dopo un’operazione *up()* su un semaforo con più processi dormienti, il semaforo sarà ancora 0, ma un processo (scelto a caso dal sistema) potrà completare il suo *down()* risvegliandosi.

I semafori risolvono il problema della perdita di wakeup, come illustrato nella Figura 2.28.

Normalmente le primitive *down()* e *up()* sono realizzate come system call.

Il sistema operativo prima di controllare il semaforo, disabilita brevemente tutti gli interrupt in modo da rendere le primitive atomiche. Nel caso ci siano più CPU occorre proteggere il semaforo con variabili di lock o istruzioni TSL (o XCHG) in modo da essere sicuri che solo una CPU alla volta possa esaminare il semaforo.

**Risolvere il problema del produttore-consumatore per mezzo dei semafori**



**Mutex**

I mutex sono una versione semplificata del semaforo. Essi sono utili solo per gestire la mutua esclusione di risorse condivise o pezzi di codice.

Sono facili da implementare ed efficienti, ciò li rende particolarmente utili nei pacchetti di thread.

Un mutex è una variabile binaria che può trovarsi in soli due stati: locked o unlocked.

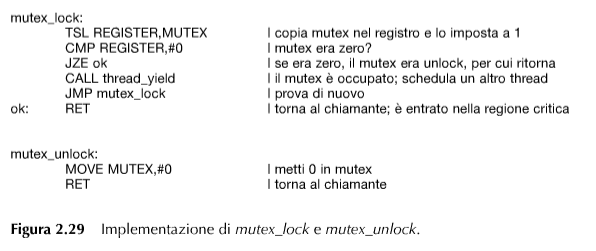
Anche se basta un bit per rappresentare un mutex, spesso si usa un numero intero: 0 significa unlocked, tutti gli altri valori significano locked.

Con i mutex si usano due procedure:

- *mutex\_lock()*: quando un processo ha bisogno di entrare nella regione critica;

- *mutex\_unlock()*: quando ha terminato l’accesso alla regione critica.

I mutex sono così semplici che se l’istruzione TSL è disponibile, possono essere realizzati nello spazio utente. Il codice di *mutex\_lock* e *mutex\_unlock* è mostrato nella Figura 2.29.



Il codice potrebbe sembrare simile a quello della Figura 2.25. Esiste però una differenza cruciale tra *enter\_region* e *mutex\_lock*. La prima crea busy waiting, la seconda invece, quando un thread fallisce nell’acquisire un lock, chiama *thread\_yield* per cedere la CPU a un altro thread, evitando il busy waiting.

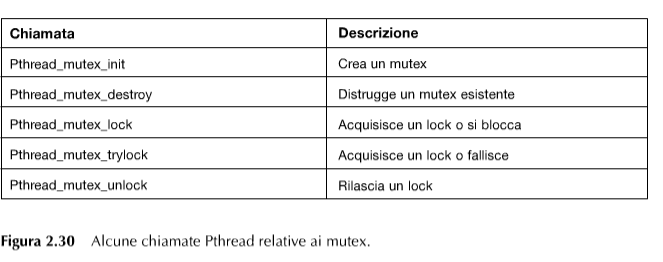
**Mutex nei Pthread**

I pthread forniscono un certo numero di funzioni utilizzabili per sincronizzare i thread. Il meccanismo di base utilizza una variabile mutex, che può essere locked o unlocked, a guardia di ogni regione critica.

Un thread che vuole entrare nella regione critica prima prova a bloccare il mutex associato. Se il mutex è unlocked, il thread può entrare immediatamente e il lock è impostato atomicamente impedendo l’ingresso ad altri thread. Se il mutex è già locked, il thread chiamante è bloccato finché non diviene unlocked.

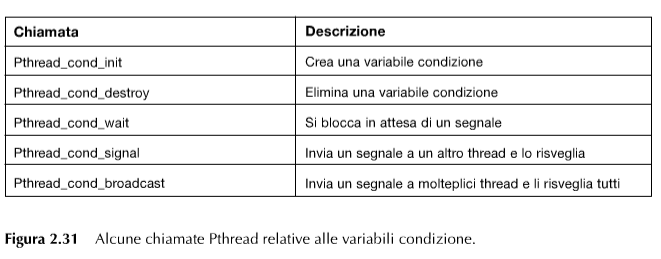
Se molteplici thread sono in attesa dello stesso mutex, quando passa in stato unlocked, solo a uno è consentito di continuare e di metterlo così a sua volta in locked. Questi lock non sono obbligatori, sta al programmatore essere certo che i thread li utilizzino correttamente.

Le principali chiamate che utilizzano i mutex sono illustrate nella Figura 2.30.



I pthread offrono un secondo meccanismo di sincronizzazione: le variabili condizione. I mutex sono buoni per permettere o meno l’accesso ad una regione critica. Le variabili condizione permettono di bloccare i processi se non si verificano alcune condizioni.

Alcune delle chiamate relative alle variabili condizione sono illustrate nella Figura 2.31.



**Monitor**

Con i semafori e mutex la comunicazione tra processi sembra apparentemente facile, invece è difficile scrivere programmi corretti.

Un monitor è un insieme di procedure, variabili e dati strutturati, raggruppate in un particolare tipo di modulo o pacchetto. I processi possono richiedere le procedure in un monitor ogni volta che vogliono, ma non possono accedere direttamente alle strutture dati interne del monitor.

I monitor hanno una proprietà importante che li rende utili per realizzare la mutua esclusione: in un dato istante, un solo processo per volta può essere attivo in un monitor.

La soluzione è l’uso delle variabili condizione abbinate alle due operazioni: *wait()* e *signal()*. Quando una procedura dentro il monitor scopre che non può continuare (es. buffer pieno o vuoto) esegue una *wait()* su una variabile condizione e blocca il processo. Un altro processo partner può risvegliare il processo inviando una *signal()* sulla variabile condizione.

Le variabili condizione non sono contatori. Non accumulano segnali per usarli in seguito come fanno i semafori.

Anche *sleep()* e *wakeup()* sembrano simili alle operazioni *wait()* e *signal()*, ma le prime avevano il difetto che mentre un processo stava tentando di entrare in sleep, l’altro stava provando a svegliarlo. In pratica, se un processo fosse stato per addormentarsi non si sarebbe accorto della sveglia dell’altro se non aveva già preso sonno. Con i monitor ciò non accade.

**Scambio di messaggi**

Questo metodo usa due primitive, *send()* e *receive()* che, come i semafori e diversamente dai monitor, sono chiamate di sistema piuttosto che costrutti del linguaggio.

- *send(destination, &message)*: invia un messaggio ad un destinatario;

*- receive(source, &message)*: riceve un messaggio da una fonte.

Se il messaggio non è disponibile, il ricevitore può rimanere bloccato finché non ne riceve uno, oppure può restituire immediatamente un codice di errore.

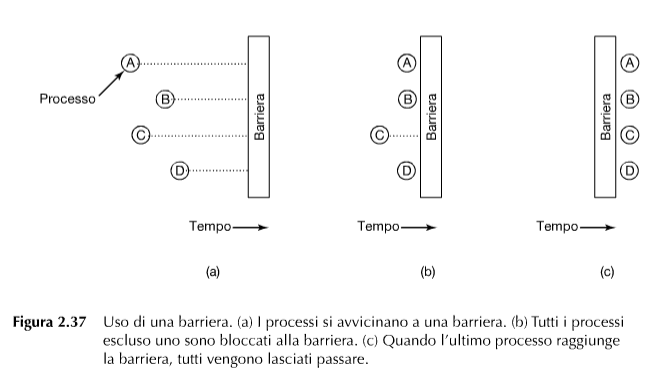
**Barriere**

Il meccanismo di sincronizzazione delle barriere è destinato a gruppi di processi, piuttosto che a comunicazioni tra due processi.

Alcune applicazioni sono suddivise in fasi e hanno la regola che nessun processo può proseguire nella fase successiva finché tutti i processi non sono pronti a procedere alla fase successiva.

Questa condotta può essere ottenuta mettendo una barriera al termine di ogni fase. Quando un processo raggiunge la barriera, viene bloccato finché tutti gli altri processi non raggiungono la barriera.

L’operazione di una barriera è mostrata nella Figura 2.37.



Nella Figura 2.37(a) vediamo quattro processi che si avvicinano a una barriera. Ciò significa che essi stanno solo elaborando e non hanno ancora raggiunto la fine della fase attuale. Un attimo dopo, il primo processo termina tutte le elaborazioni richieste durante la prima fase. Esegue poi la primitiva *barrier()*. Il processo è quindi sospeso. Un attimo dopo, un secondo e un terzo processo terminano la prima fase ed eseguono anche loro la primitiva *barrier()*. Questa situazione è illustrata nella Figura 2.37(b). Alla fine, quando anche l’ultimo processo C raggiunge la barriera, i processi sono rilasciati, come mostrato nella Figura 2.37(c).

**Scheduling**

Quando un computer è multiprogrammato, ha più processi o thread che competono per ottenere la CPU nel medesimo istante. Questa situazione si verifica quando due o più di loro sono in stato pronto. Con una CPU disponibile un solo processo/thread può essere selezionato per entrare in esecuzione. La parte del sistema operativo che effettua questa scelta è chiamato scheduler e l’algoritmo utilizzato è detto algoritmo di scheduling.

Molti dei problemi che si applicano allo scheduling dei processi si applicano anche ai thread. Quando il kernel gestisce i thread, lo scheduling è fatto per thread indipendentemente dal processo di appartenenza.

**Introduzione allo scheduling**

Ai tempi dei sistemi batch l’algoritmo di scheduling era semplice: bastava eseguire il prossimo job sul nastro.

Con i sistemi multiprogrammati, l’algoritmo di scheduling è diventato più complesso in quanto ci sono più utenti in attesa di servizio.

Con l’avvento del PC, la situazione si è evoluta in due modi.

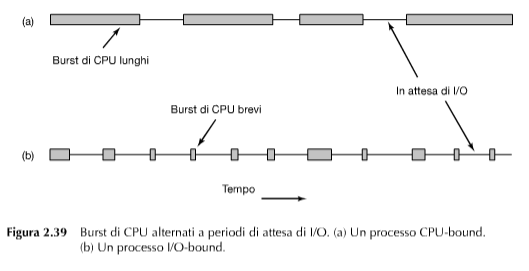
Primo, per la maggior parte del tempo è attivo un solo processo. Un utente che sta utilizzando un word processor molto probabilmente non sta utilizzando concorrentemente un’altra applicazione. Quindi quando un utente digita un comando nel programma, per lo scheduler è semplice individuare il processo da eseguire, il programma di word processing è l’unico candidato.

Secondo,oggi i computer sono diventati così veloci che raramente la CPU è una risorsa scarsa. La maggior parte dei programmi per PC è limitata dalla velocità di inserimento dati dell’utente.

La situazione è completamente diversa se si considerano i server collegati in rete.

**Comportamento dei processi**

Quasi tutti i processi alternano passi di elaborazione a richieste di I/O, come illustrato nella Figura 2.39.



Alcuni processi, come quelli della Figura 2.39(a), spendono la maggior parte del loro tempo in elaborazione, mentre altri, come quelli della Figura 2.39(b), spendono la maggior parte del loro tempo in attesa dell’I/O.

La prima situazione è chiamata CPU-bound, la seconda I/O-bound.

I processi CPU-bound hanno burst di CPU lunghi e attese di I/O poco frequenti, mentre i processi I/O-bound hanno burst di CPU brevi e attese di I/O frequenti.

Si noti che con l’incremento delle performance delle CPU i processi tendono ad essere più I/O-bound che CPU-bound.

**Quando effettuare lo scheduling**

Un aspetto cruciale dello scheduling è stabilire quando è il momento opportuno per eseguirlo. Le situazioni in cui è necessario schedulare sono varie:

- quando è creato un nuovo processo si deve decidere se deve essere eseguito il processo genitore o il processo figlio;

- quando un processo termina, qualche altro processo deve essere scelto nell’insieme dei processi pronti;

- quando un processo si blocca sull’I/O o su un semaforo bisogna selezionare un altro processo dall’elenco dei processi pronti;

- quando si verifica un interrupt di I/O, se l’interrupt proveniva da un dispositivo di I/O che adesso ha terminato il lavoro, qualche processo bloccato in attesa del dispositivo di I/O può ora essere pronto a partire. Sta allo scheduler stabilire se eseguire il processo appena diventato pronto, quello che era in esecuzione al momento dell’interrupt o qualcun altro.

**Tipi di scheduling**

Gli algoritmi di scheduling possono essere suddivisi in due categorie rispetto alla capacità dello scheduling di interrompere l’esecuzione dei processi:

- non preemptive: lo scheduler una volta mandato in esecuzione un processo lo lascia andare finché non si blocca (attesa di I/O o di un altro processo) o rilascia spontaneamente la CPU;

- preemptive: lo scheduler sceglie un processo da eseguire per un tempo massimo stabilito. Se il processo è ancora in esecuzione al termine dell’intervallo di tempo, lo scheduler lo sospende e sceglie un altro processo da eseguire (se disponibile).

**Categorie di algoritmi di scheduling**

In diverse aree applicative sono necessari differenti sistemi operativi e, conseguentemente, specifici algoritmi di scheduling in quanto quello che lo scheduler dovrebbe ottimizzare non è medesima cosa in tutti i sistemi.

Saranno trattati tre ambienti differenti:

- sistemi batch;

- sistemi interattivi;

- sistemi real-time.

I sistemi batch sono ancora utilizzati nel mondo degli affari per l’elaborazione di paghe, esecuzione di accrediti e addebiti su conti corrente ed altre attività periodiche.

Nei sistemi batch non ci sono utenti impazienti in attesa al loro terminale di una risposta rapida a una piccola richiesta. Di conseguenza sono spesso accettabili algoritmi non preemptive.

In un ambiente con utenti interattivi, l’uso della preemptive è essenziale per evitare che un processo monopolizzi la CPU e neghi il servizio agli altri.

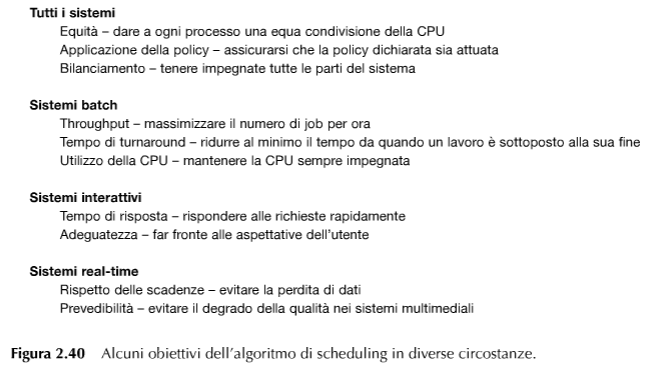
Anche i server ricadono in questa categoria, dato che normalmente servono molteplici utenti (remoti), e tutti sempre di fretta.

In sistemi con vincoli real-time, la preemption potrebbe non essere necessaria poiché i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e generalmente fanno il loro lavoro e si bloccano rapidamente.

**Obiettivi degli algoritmi di scheduling**

Per progettare un algoritmo di scheduling è necessario sapere che cosa dovrebbe fare un buon algoritmo. Alcuni obiettivi dipendono dall’ambiente, mentre altri sono auspicabili in tutti i casi.

Alcuni obiettivi sono elencati nella Figura 2.40.



In ogni situazione, l’equità è importante. Processi comparabili dovrebbero avere una congrua parte della CPU.

Forzare le policy del sistema è scorretto. Se la policy locale è che i processi di controllo della sicurezza devono essere eseguiti quando lo richiedono, anche se ciò implica che le paghe siano in ritardo di 30 secondi, lo scheduler deve essere certo che questa policy venga rispettata.

Tutte le parti del sistema devono essere impegnate in modo bilanciato. Se la CPU e i dispositivi di I/O sono tenuti sempre impegnati vengono eseguiti più lavori al secondo rispetto ad avere componenti inattivi.

Nei sistemi batch sono molto rilevanti due metriche:

- il throughput, cioè il numero di job completati nell’unità di tempo;

- il tempo di turnaround, cioè il tempo medio di esecuzione dei processi batch.

Anche se è una metrica non appropriata per i sistemi batch, viene spesso utilizzata la percentuale di utilizzo della CPU.

Nei sistemi interattivi è fondamentale il tempo di risposta: il tempo che trascorre dall’invio del comando a quando si ottiene il risultato.

L’adeguatezza della risposta è, invece, la percezione degli utenti rispetto al tempo impiegato per svolgere quel compito. Lo scheduler potrebbe assegnare male le priorità di esecuzione e causa un degrado delle risposte.

Per i sistemi in tempo reale è essenziale soddisfare le scadenze ed evitare errori durante l’esecuzione del processo.

**Scheduling nei sistemi batch**

Gli algoritmi di scheduling utilizzati nei sistemi batch sono:

- first-come first-served;

- shortest job first;

- shortest remaining time next.

Tra questi alcuni sono validi anche nei sistemi interattivi.

**First-come first-served**

L’algoritmo first-come first-served è il più semplice algoritmo di scheduling non preemptive.

Con questo algoritmo i processi sono assegnati alla CPU nell’ordine in cui la richiedono. Fondamentalmente c’è una singola coda dei processi in stato pronto.

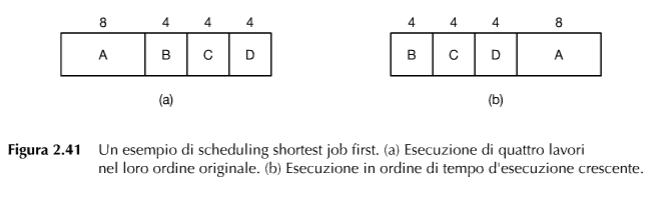
Questo algoritmo è facile da capire e da programmare. D’altro canto, presenta il seguente problema: un sistema con processi eterogenei per tempo di esecuzione potrebbe rallentare i processi veloci in assenza di preemption.

**Shortest job first**

Si tratta di un algoritmo senza preemption in cui i tempi di esecuzione dei processi sono noti in anticipo. La predizione dei tempi di esecuzione per job ripetitivi non è difficile.

I job entrano nella coda in ordine, ma lo scheduler sceglie quello che termina prima.

Si osservi la Figura 2.41.



Vi troviamo i lavori A, B, C e D con un tempo di esecuzione di 8, 4, 4 e 4 minuti, rispettivamente. Eseguendoli in ordine il tempo di turnaround sarebbe di 8 minuti per A, 12 minuti per B, 16 per C e 20 per D, con una media di 14 minuti.

Consideriamo questi lavori usando l’algoritmo shortest job first, come illustrato nella Figura 2.41(b). I tempi di turnaround diventano 4, 8, 12 e 20 minuti, con una media di 11 minuti.

La formula di turnaround medio è *(4a+3b+2c+d)/4*. Essa evidenzia che *a* influenza la media più degli altri, così dovrebbe essere il lavoro più breve, con *b* come successivo e così via.

Si noti che l’algoritmo in questione è ottimale solo nel caso in cui tutti i lavori siano disponibili contemporaneamente.

**Shortest remaining time next**

Questo algoritmo è una versione preemptive dell’algoritmo shortest job first.

Lo scheduler in questo caso sceglie sempre il processo il cui tempo che manca alla fine dell’esecuzione è il più breve. Anche in questo caso il tempo di esecuzione deve essere conosciuto in anticipo. All’arrivo di un nuovo lavoro il suo tempo totale è confrontato al tempo restante dei processi attuali. Se il nuovo lavoro ha bisogno di minor tempo del processo attuale per terminare, il processo attuale viene sospeso ed è avviato il nuovo lavoro.

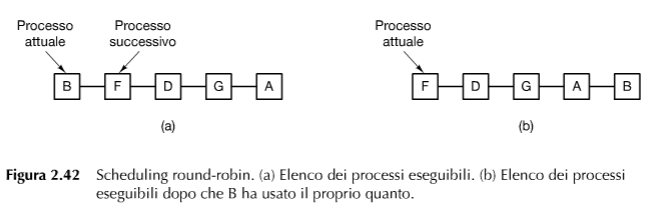
Questo schema permette ai nuovi lavori brevi di ottenere un buon servizio.

**Scheduling nei sistemi interattivi**

**Scheduling round-robin**

Il round-robin è uno degli algoritmi più vecchi, semplice, equilibrato e diffuso.

Ad ogni processo è assegnato un quanto di tempo durante il quale gli è consentito di essere eseguito. Se alla fine del quanto il processo è ancora in esecuzione, la CPU viene prelazionata e assegnata a un altro processo. Se il processo si blocca o termina prima che sia trascorso il quanto, viene fatto uno scambio di CPU quando il processo si blocca. Il round-robin è facile da realizzare. Tutto quello che deve fare lo scheduler è mantenere una lista dei processi eseguibili, come illustrato nella Figura 2.42(a). Quando il processo esaurisce il suo quanto, è messo alla fine dell’elenco, come illustrato nella Figura 2.42(b).



In questo algoritmo è cruciale la scelta del quanto di tempo. Un valore troppo breve provoca troppe interruzioni dei processi causando overhead; d’altra parte, un valore troppo alto provoca una coda di attesa eccessiva.

**Scheduling a priorità**

Nello scheduling round-robin c’è l’assunzione implicita che tutti i processi sono ugualmente importanti. In molti contesti reali questa è un’ipotesi troppo restrittiva.

Ciò porta allo scheduling a priorità.

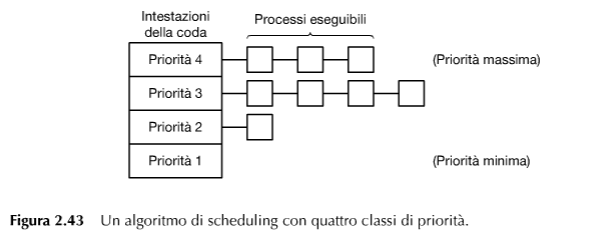
L’idea di base è assegnare a ciascun processo una priorità e il processo eseguibile con la priorità più alta è quello cui è consentita l’esecuzione.

Per evitare che i processi ad alta priorità siano eseguiti in tempi indeterminati, lo scheduler può abbassare la priorità del processo attualmente in esecuzione a ogni scatto del clock. Se questa azione fa sì che la sua priorità vada al di sotto di quella del processo successivo, avviene uno scambio di processo. In alternativa a ciascun processo può essere assegnato un quanto di tempo massimo in cui può essere eseguito.

Le priorità possono essere assegnate ai processi staticamente o dinamicamente.

È conveniente dividere i processi per classi di priorità utilizzando lo scheduling con priorità per le classi e quello round-robin all’interno di ciascuna classe.

La Figura 2.43 illustra un sistema con quattro classi di priorità.



**Shortest process next**

Poiché l’algoritmo shortest job first produce sempre il minor tempo medio di risposta sui sistemi batch, sarebbe apprezzabile che si potesse usare anche sui sistemi interattivi.

Il problema sta nel calcolare il tempo di esecuzione che non è sempre lo stesso.

Un possibile approccio è quello di fare delle stime basate sull’esperienza maturata ad eseguire il processo con il tempo di esecuzione stimato più breve.

Supponiamo che il tempo stimato per comando per un certo terminale sia *T0*. Supponiamo che la sua esecuzione successiva sia misurata come *T1*. Potremmo aggiornare la nostra stima facendo una somma pesata dei numeri, ossia *aT0 + (1 - a)T1*. Tramite la scelta di *a* possiamo decidere se avere il processo di stima che dimentica in breve tempo le ultime esecuzioni o le ricorda a lungo.

Con *a = ½* abbiamo le seguenti stime:

*T0, T0/2 + T1*/2, *T0/4 + T1*/4 *+ T2/2*, *T0/8 + T1*/8 *+ T2/4 + T3/2.*

Dopo tre esecuzioni, il peso di *T0* nella nuova stima è sceso a *⅛*.

Si tenga anche conto che le misure più recenti sono più attendibili rispetto a quelle passate (aging).

Praticamente stiamo dicendo che più un processo si trova in scheduling meno tempo gli viene concesso dopo ogni esecuzione, facendo in modo di concedere più cpu ai nuovi processi

**Scheduling garantito**

Un approccio completamente diverso per lo scheduling è quello di fare promesse reali agli utenti riguardo alle prestazioni e poi mantenerle.

Una promessa che è realistico fare e semplice mantenere è la seguente: se ci sono *n* utenti collegati mentre state lavorando, avrete all’incirca *1/n* del tempo della CPU.

Per mantenere questa promessa, il sistema deve tener traccia di quanta CPU ha ricevuto ogni processo dal momento della sua creazione. Poi calcola la quantità di CPU che ognuno ha diritto ad avere, cioè il tempo dalla creazione diviso *n*. Se un qualsiasi processo ha un valore più basso di tale rapporto è il successivo candidato ad essere messo in esecuzione, finché il suo rapporto è il più basso tra tutti quelli pronti.

**Scheduling a lotteria**

Fare promesse agli utenti e poi mantenerle è una bella idea, ma difficile da realizzare.

Si può realizzare un algoritmo più semplice, lo scheduling a lotteria.

L’idea è quella di assegnare ai processi un biglietto della lotteria per le diverse risorse del sistema, come il tempo di CPU. Ogni volta che deve essere presa una decisione di scheduling si pesca un biglietto della lotteria e il processo che ha quel biglietto si aggiudica la risorsa.

Differentemente dallo scheduling con le priorità in cui è difficile dire cosa significa avere una certa priorità, in questo algoritmo se un processo possiede 20 biglietti su 100 ha il 20% delle probabilità di ottenere la risorsa (che al crescere del tempo approssima la frequenza).

Lo scheduling a lotteria ha alcune proprietà interessanti. Per prima cosa, è reattivo. Infatti, anche i processi neonati possono vincere la lotteria, fin dalle prime scelte dello scheduler. In secondo luogo, i processi che cooperano, se lo desiderano, possono scambiarsi i biglietti che detengono per alterare le priorità di esecuzione.

Lo scheduling a lotteria può essere utilizzato per risolvere problemi difficili da gestire con altri metodi. Ad esempio, si consideri lo streaming video, dove sono necessarie differenti velocità di trasferimento (espresse in frame al secondo) a seconda del processo. Assegnando tanti biglietti quanti sono i frame automaticamente la CPU approssima le proporzioni desiderate.

**Scheduling fair-share**

Finora abbiamo supposto che ogni processo fosse schedulato per proprio conto, senza considerare a chi appartenesse.

Di conseguenza, se un utente X avvia 4 processi (A, B, C e D), e l’utente Y ne avvia uno solo (E), l’utente X si prenderà l’80% della CPU e l’utente Y solo il 20%.

Per evitare questo, prima di schedularlo, alcuni sistemi prendono in considerazione chi possiede un processo. In questo modello, a ogni utente viene assegnata una frazione di CPU (quindi non in base ai processi) e lo scheduler raccoglie i processi in modo tale da farla rispettare.

50% della CPU a ogni utente avremo: A E B E C E D E A E B E C E D E …

75% della CPU all’utente X e il 25% all’utente Y avremo: A B E C D E A B E C D E ...

**Scheduling nei sistemi real-time**

In un sistema real-time il tempo gioca un ruolo essenziale.

I sistemi real-time sono suddivisi in due categorie:

- hard real-time: le scadenze devono essere sempre rispettate;

- soft real-time: il mancato rispetto di una scadenza non è auspicabile, ma comunque tollerabile.

Gli eventi in un sistema real-time possono essere classificati come:

- periodici, quando si verificano a intervalli di tempo regolari;

- non periodici, quando si verificano in modo imprevedibile.

Un sistema soft real-time con eventi periodici è sostenibile se riesce a far fronte agli eventi stessi, ovvero se riesce a trattare un evento prima che ne arrivi un altro.

Se ci sono *m* eventi periodici e ogni evento avviene con frequenza pari a *1/P*, supponendo che ciascun evento richieda *C* secondo di tempo CPU per gestirlo, allora il sistema è in grado di reggere il carico se e solo se: .

Un sistema real-time che soddisfa questo criterio è detto schedulabile.

**Scheduling dinamico e statico**

Gli algoritmi di scheduling per i sistemi real-time possono essere statici o dinamici.

Gli algoritmi statici prendono la decisione di scheduling prima che il sistema inizi a funzionare. Questa tecnica è applicabile solo quando ci sono informazioni disponibili in anticipo riguardo il lavoro da svolgere e le scadenze da rispettare.

Gli algoritmi dinamici prendono le decisioni di scheduling in fase di esecuzione. Questa tecnica non necessita di conoscere in anticipo alcuna informazione sul compito da svolgere e sui tempi.

**La policy in contrapposizione al meccanismo**

Finora abbiamo presupposto che tutti i processi del sistema appartengano a utenti differenti e che siano di conseguenza in competizione per la CPU. Talvolta, accade che un processo abbia molti figli eseguiti sotto il suo controllo (es. un dbms può avere molti processi figli).

Il processo principale sa quale dei suoi figli sia il più importante (o critico dal punto di vista del tempo) e quale meno. Sfortunatamente nessuno degli scheduler finora presentati accetta alcun input dai processi utente riguardo alle decisioni di scheduling. Di conseguenza, lo scheduler raramente prende la decisione migliore.

La soluzione a questo problema sta nel separare il meccanismo di scheduling dalla politica di scheduling. L’algoritmo di scheduling è quindi in qualche modo parametrizzato, ma i parametri possono essere riempiti dai processi utente.

**Scheduling a thread**

Quando i processi hanno più thread, possiamo definire due livelli di parallelismo: sui processi e sui thread. Lo scheduling in questi ambienti si differenzia a seconda che siano supportati thread utente o a livello kernel (o entrambi).

Consideriamo i thread utente. Dato che il kernel non è a conoscenza dell’esistenza dei thread, opera come fa sempre, prendendo un processo, diciamo A, e assegnando ad A il controllo per il suo quanto. Ora lo scheduler di thread interno ad A decide quale thread eseguire, diciamo A1. Poiché per questi thread non ci sono interrupt esso può continuare a funzionare quanto vuole. Se utilizza l’intero quanto del processo, il kernel seleziona un altro processo da eseguire.

Quando verrà rieseguito il processo A, il thread A1 riprenderà l’esecuzione. Continuerà a consumare tutto il tempo di A finché non è finito. Il suo comportamento non influisce su altri processi.

La sequenza di attivazione sarà: *A1 B1 A2 B2 A3 B3*. Questo non è possibile con questi parametri e i thread utente.

Nel caso dei thread a livello kernel, quest’ultimo conosce i thread e ne prende uno per l’esecuzione. Al thread è assegnato un quanto di tempo e viene forzatamente sospeso se supera quel valore di tempo di esecuzione.

La sequenza di attivazione *A1 B1 A2 B2 A3 B3* è ora possibile.

Vantaggi e svantaggi dei thread a livello utente:

- lo scambio di esecuzione tra thread è veloce e sono sufficienti poche istruzioni macchina;

- se un thread va in blocco su una operazione di I/O si deve sospendere l’intero processo;

- si può utilizzare lo scheduler di thread specifico dell’applicazione (es. web server) e che abiliti una strategia di attivazione migliore del kernel, poiché conosce ciò che fanno i vari thread.

Vantaggi e svantaggi dei thread a livello kernel:

- lo scambio di contesto è oneroso di molti ordini di grandezza rispetto all’altro caso, poiché occorre invalidare la mappa di memoria e la cache;

- il kernel può decidere qual thread mandare in esecuzione tenendo conto anche dell’overhead causato da un eventuale cambio di contesto;

- se un thread va in blocco non sospende l’intero processo.

Questo non vuol dire che non esista più il problema della mutua esclusione in quanto il kernel dice solo quando eseguirli e **non come accedere alla memoria** in quanto i thread hanno la memoria condivisa.